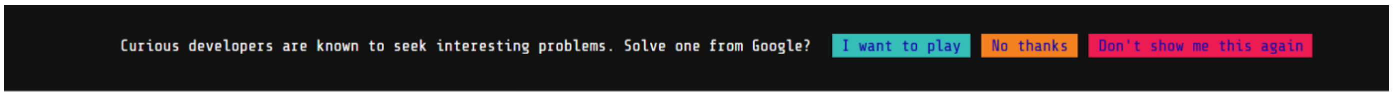
Fabian Bosler Follow
Jun 30, 2021 · 20 min read ★ · Listen

Google Has a Secret Hiring Challenge Called Foobar

Here's how I got in and why I loved the problems



```
Curious developers are known to seek interesting problems. Solve one from Google?  I want to play  No thanks  Don't show me this again
```

Invitational message for Google Foobar

This article outlines my experience with Google's Foobar: how I got the invitation, the problems, my submissions, and what happened after. Be advised that this article contains a hefty amount of spoilers.

Intro

The first time I read about Google's Foobar challenge must have been sometime in 2019. I had been on Codewars occasionally and enjoyed solving algorithmic problems, so when I read about Foobar, I wanted to give it a try. But first, let's take a step back and summarize what Foobar is.

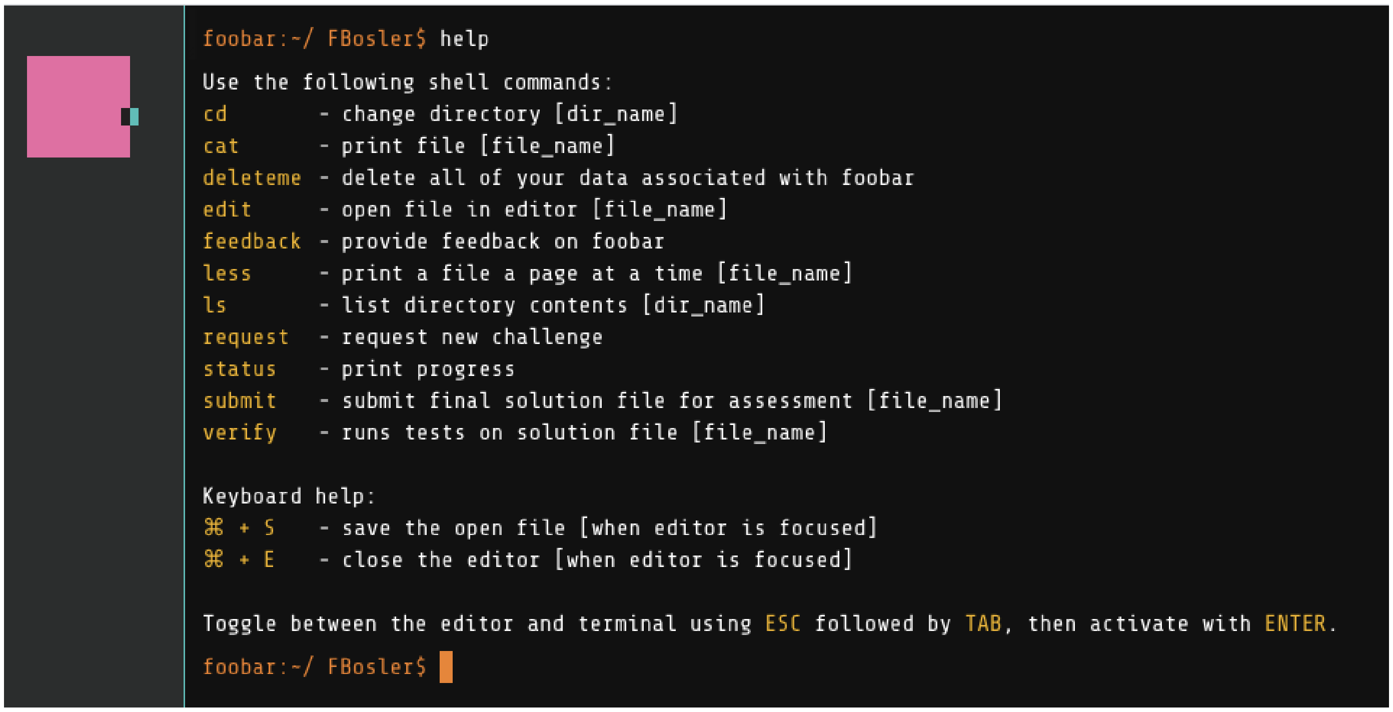
What is Foobar?

Foobar is a semi-secret hiring challenge Google put out there to find talented developers

- **Level 1:** One Question — 48 hours
- **Level 2:** Two Questions — 72 hours for each question
- **Level 3:** Three Questions — 7 days for each question
- **Level 4:** Two Questions — 2 weeks for each question
- **Level 5:** One Question — 22 days

For every problem, you can either submit a solution written in Python (2.7) or Java. Before submitting, you can also validate your solution and run it against a suite of secret tests. You aren't told the test cases, though, only if your algorithm passes the test or not.

What is cool about Foobar is that it takes place on a virtual server, to which you gain access once invited into the challenge.



```
foobar:~/ FBosler$ help
Use the following shell commands:
cd        - change directory [dir_name]
cat       - print file [file_name]
deleteme  - delete all of your data associated with foobar
edit      - open file in editor [file_name]
feedback  - provide feedback on foobar
less      - print a file a page at a time [file_name]
ls        - list directory contents [dir_name]
request   - request new challenge
status    - print progress
submit    - submit final solution file for assessment [file_name]
verify    - runs tests on solution file [file_name]

Keyboard help:
⌘ + S     - save the open file [when editor is focused]
⌘ + E     - close the editor [when editor is focused]

Toggle between the editor and terminal using ESC followed by TAB, then activate with ENTER.
foobar:~/ FBosler$
```

Foobar: Available commands

The scenario is set in a fictitious spaceship run by commander Lambda, a pretty mean

The trick part about Foobar really is getting in. There are two ways of getting in: one is being invited by Google directly while surfing the internet, and the other is via an invitational code.

A. Direct Invitation

This way into Foobar is much cooler because it typically catches the recipient by surprise. What typically happens is that a person keeps googling (advanced) software engineering topics. Once Google is confident enough that the person knows what they are looking for, the search bar unravels, and a challenge is offered.

B. Invitation Link

All participants in Google's Foobar challenge are given a personal invitation link after completing level 3 and then after level 5. The participants can use the invitation links to invite friends/coworkers.

So how did I get into Foobar?

Sometime in 2019, after reading about Foobar for the first time, I tried to google the most obscure technical questions with the intent of triggering the invitation. That didn't work. Nothing happened. So I kind of forgot about it until about a year ago when I was researching the following article:

Reimplementing popular Python functionalities in JavaScript

Learning how to cope in a new world and making the best use of the tools available.

towardsdatascience.com

When the search bar unraveled and I was asked if I was up for a challenge, I knew exactly what was up and hit yes. Embarrassingly enough, I was mostly googling things like `list comprehension` and `dictionary comprehension` at the time.

the levels. I contemplated not posting the article because I didn't want to "ruin" the challenge by making the solutions publicly accessible. However, they are anyway. A couple of people have already written about the problems, and the solutions can be found on the internet. Additionally, I had a lot of fun going through the problems and realized that some of the algorithms could actually be applied to real-life problems I had before. So I felt the challenges are not only very entertaining to read about but also quite informative.

You can find the problems and solutions here. In the following part, I am going to describe my thought process and use of real-life applications for each problem.

Problem 1.

The first problem asks you to do the following:

```
Problem:

Write a function called solution(data, n) that takes in a list of less
than 100 integers and a number n, and returns that same list but with
all of the numbers that occur more than n times removed entirely
```

This one took a couple of minutes to figure out, and the problem didn't really force me to learn something particularly new:

```
Solution:

from collections import Counter

def solution(data, n):
    counted = Counter(data)
    return [k for k, v in counted.items() if v <= n]
```

We can use `collections.Counter` to find the solution. My actual submission was a bit more

```
Problem:

Given a non-empty list of positive integers l and a target positive
integer t, write a function solution(l, t) which verifies if there is
at least one consecutive sequence of positive integers within the list
l (i.e. a contiguous sub-list) that can be summed up to the given
target positive integer t (the key) and returns the lexicographically
smallest list containing the smallest start and end indexes where this
sequence can be found, or returns the array [-1, -1] in the case that
there is no such sequence (to throw off Lambda's spies, not all number
broadcasts will contain a coded message).
```

Interesting, but not too complicated to solve (I think). I looped over the values and, for each value, checked if there is a sub-list starting with the value that adds up to `t`. Complexity is at worst somewhere around `(n^2)/2 — n/2` which certainly isn't great, but I didn't feel like performance was an issue, so I left it as is. Find the full problem and solution here.

Problem 2b.

```
Problem:

Oh no! Commander Lambda's latest experiment to improve the efficiency
of her LAMBCHOP doomsday device has backfired spectacularly. She had
been improving the structure of the ion flux converter tree, but
something went terribly wrong and the flux chains exploded. Some of
the ion flux converters survived the explosion intact, but others had
their position labels blasted off. She's having her henchmen rebuild
the ion flux converter tree by hand, but you think you can do it much
more quickly - quickly enough, perhaps, to earn a promotion!

Flux chains require perfect binary trees, so Lambda's design arranged
the ion flux converters to form one. To label them, she performed a
post-order traversal of the t[illegible]ters and labeled each
converter with the order of [illegible] in the traversal, starting
at 1. For example, a tree of 7 converters would look like the
following:
```

```
Write a function solution(h, q) - where h is the height of the perfect
tree of converters and q is a list of positive integers representing
different flux converters - which returns a list of integers p where
each element in p is the label of the converter that sits on top of
the respective converter in q, or -1 if there is no such converter.
For example, solution(3, [1, 4, 7]) would return the converters above
the converters at indexes 1, 4, and 7 in a perfect binary tree of
height 3, which is [3, 6, -1].
```

This was the first problem I really liked because it got me thinking about how such trees would generalize and which information would be available and easily computable.

The first thing we need to find for a tree of height `h` is the value of the root node, which is just the number of values in the tree, in other words:

```
base_root = sum([2 ** level for level in range(h)])
```

We need a function that finds the root for every given number finds the root. This function would require our `base_root` the `height` of the tree and obviously, the `element` to check. The function would work then.

1. Check if we are out of bounds (i.e., no node on top) `-> -1`
2. Calculate `left_leaf = root_value — 2**(height-1)` and `right_leaf = root_value -1`
3. Check that `element not in [left_leaf, right_leaf]` `else -> root_value`
4. Now while `element` is neither of those two leaves we traverse the branch of the tree where the element is going to sit (sort of like a binary search). If `element` is less than `left_leaf` it's going to be in the left side of the tree; otherwise, in the right side of the tree.
5. Set `root_value` to either `left_leaf` or `right_leaf` and reduce the height by one.
6. Rinse and repeat. We can do this recursively or use a while-loop.

This one was a real nailbiter. But also the first one I really enjoyed as it had me restudy Markov Chains. Also, whilst working on the problem, I realized that I came across a couple of real-life applications where the theory would have been helpful. The first one that came to mind is customer lifetime value in a situation where customers can move from one segment to another either implicitly (as indicated by their buying behavior) or explicitly (as indicated by a change in subscription plan).

Now, if one can calculate the expected terminal distribution (on segments) of a new customer and if one knew the approximate lifetime value of a customer in a particular segment, one would have a pretty accurate estimation of the lifetime value of a random new customer.

Anyway, let's move to the problem:

```
Problem:

Making fuel for the LAMBCHOP's reactor core is a tricky process
because of the exotic matter involved. It starts as raw ore, then
during processing, begins randomly changing between forms, eventually
reaching a stable form. There may be multiple stable forms that a
sample could ultimately reach, not all of which are useful as fuel.

Commander Lambda has tasked you to help the scientists increase fuel
creation efficiency by predicting the end state of a given ore sample.
You have carefully studied the different structures that the ore can
take and which transitions it undergoes. It appears that, while
random, the probability of each structure transforming is fixed. That
is, each time the ore is in 1 state, it has the same probabilities of
entering the next state (which might be the same state).  You have
recorded the observed transitions in a matrix. The others in the lab
have hypothesized more exotic forms that the ore can become, but you
haven't seen all of them.

Write a function solution(m) that takes an array of array of
nonnegative ints representing how many times that state has gone to
the next state and return an array of ints for each terminal state
giving the exact probabilities of each terminal state, represented as
the numerator for each state, then the denominator for all of them at
the end and in simplest form. The matrix is at most 10 by 10. It is
```

```
For example, consider the matrix m:
```
[
  [0,1,0,0,0,1], # s0, the initial state, goes to s1 and s5 with
equal probability
  [4,0,0,3,2,0], # s1 can become s0, s3, or s4, but with different
probabilities
  [0,0,0,0,0,0], # s2 is terminal, and unreachable (never observed in
practice)
  [0,0,0,0,0,0], # s3 is terminal
  [0,0,0,0,0,0], # s4 is terminal
  [0,0,0,0,0,0], # s5 is terminal
]
```
So, we can consider different paths to terminal states, such as:
```
s0 -> s1 -> s3
s0 -> s1 -> s0 -> s1 -> s0 -> s1 -> s4
s0 -> s1 -> s0 -> s5
```
Tracing the probabilities of each, we find that
s2 has probability 0
s3 has probability 3/14
s4 has probability 1/7
s5 has probability 9/14
So, putting that together, and making a common denominator, gives an
answer in the form of
```
[s2.numerator, s3.numerator, s4.numerator, s5.numerator, denominator]
which is
[0, 3, 2, 9, 14].
```
```

We actually need quite a bit of linear algebra to solve this, which we have to implement ourselves because there is no linear algebra in Standard Python 2.7 libraries. You can find the full solution in the repo, but I'll walk through the steps on a high level. First, though, you should check out chapter 11.2 (page 426 onward) of this amazing lecture of Dartmouth, where all the theoretical foundations and their requirements are explained beautifully. The fundamental takeaway is that we are looking for `terminal_states` that are computed as:

The lecture explains how we can get `Q` and `R` but in essence, those are just parts of the matrix after some sorting and normalization. It's a bit tedious to reimplement Gaussian elimination, which we need for the inversion, but it's ultimately pretty straightforward once the theoretical foundations are understood.

You can find the full problem and solution here.

Problem 3b.

The problem is a shortest-path problem with the additional condition that one obstacle can be ignored. A problem like this translates easily into a real-life application like finding the shortest route from one city to another, only paying a certain amount of highway tolls. Or to pass at most one McDonalds because you don't want your children to be too upset :)

```
Problem:

...

Unfortunately (again), you can't just remove all obstacles between the
bunnies and the escape pods -
at most you can remove one wall per escape pod path, both to maintain
structural integrity of the station and
to avoid arousing Commander Lambda's suspicions.

You have maps of parts of the space station, each starting at a prison
exit and ending at the door to an escape pod.
The map is represented as a matrix of 0s and 1s, where 0s are passable
space and 1s are impassable walls.
The door out of the prison is at the top left (0,0) and the door into
an escape pod is at the bottom right (w-1,h-1).

Write a function solution(map) that generates the length of the
shortest path from the prison door to the escape pod,
where you are allowed to remove one wall as part of your remodeling
plans. The path length is the total number of nodes
you pass through, counting both the entrance and exit nodes. The
starting and ending positions are always passable (0).
The map will always be solvable, though you may or may not need to
remove a wall.
The height and width of the map can be from 2 to 20. Moves can only be
```

To solve the problem, I used a breadth-first search on what I called a `double layered graph`. I am sure I didn't invent this and just read about it somewhere, but the final implementation felt really good. Especially after initially trying a brute-force approach.

The way I thought about the problem was to create two identical layers of the maze and put them on top of each other. While walking through the maze, we discover adjacent nodes and put them into a double-ended queue. We then remove (`visit them`) the nodes in our queue one by one and discover new nodes. In the base layer, walls can be discovered as visitable nodes. However, walls in the shadow layer can not be discovered (i.e., they can not be passed through).

When visiting a node in the base layer, we mark the node in the base layer and the corresponding node in the shadow layer as visited. We do this because it's strictly better to discover a node without hitting a wall first. Once we step into a wall, we transition to the shadow layer and thus making sure that this particular traversal will not hit another wall.

You can find the full problem and solution here.

Problem 3c.

This problem was much easier than 3a. and 3b. It's a fun problem that becomes easy to solve once we consider binary representation.

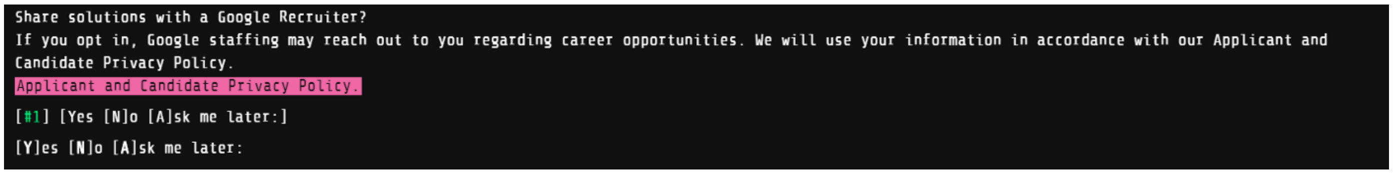
```
Problem:

...

The fuel control mechanisms have three operations:

1) Add one fuel pellet
2) Remove one fuel pellet
3) Divide the entire group of fuel pellets by 2 (due to the
destructive energy released when a quantum antimatter
pellet is cut in half, the safety controls will only allow this to
happen if there is an even number of pellets)
```

```
number up to 309 digits long, so there won't ever be more pellets than
you can express in that many digits.
```

Wow, a number that is 309 digits long is big. That is somewhere around 2 to the power of 1026. This is a situation where we are really entering `OverflowError` territory and just naively applying trial-and-error would result in a horribly inefficient algorithm.

However, once we understand that by converting `n` to binary and looking at the number of trailing 0's, we already know how many times we can divide by 2. Removing those zeros is equivalent to dividing by 2 to the power of the number of zeros. If there are no trailing zeros, we check for the next loop if adding or removing one leads to more trailing zeros (i.e., fewer operations).

You can find the full problem and solution here.



```
Share solutions with a Google Recruiter?
If you opt in, Google staffing may reach out to you regarding career opportunities. We will use your information in accordance with our Applicant and
Candidate Privacy Policy.
Applicant and Candidate Privacy Policy.
[#1] [Yes [N]o [A]sk me later:]
[Y]es [N]o [A]sk me later:
```

Recruiter policy after level 3

Problem 4a.

This was the problem I spent by far the most time on if I remember correctly. The problem boils down to finding a maximum matching on a graph. The algorithm I first wanted to use was the Blossom Algorithm. However, Blossom has a complexity of `O(|E||V|^2)` , there is a variation of the Blossom Algorithm with a complexity of `O(|E||V|^(1/2))` . Silvio Micali and Vijay Vazirani developed this algorithm. A proof can be found here.

I dug a little into the algorithm and even watched a YouTube talk about it, but I did not feel like I really had understood the algorithm, so I decided against using it. Ultimately, I

```
...

You will set up simultaneous thumb wrestling matches. In each match,
two guards will pair off to thumb wrestle. The guard with fewer
bananas will bet all their bananas, and the other guard will match the
bet. The winner will receive all of the bet bananas. You don't pair
off guards with the same number of bananas (you will see why,
shortly). You know enough guard psychology to know that the one who
has more bananas always gets over-confident and loses. Once a match
begins, the pair of guards will continue to thumb wrestle and exchange
bananas, until both of them have the same number of bananas. Once that
happens, both of them will lose interest and go back to guarding the
prisoners, and you don't want THAT to happen!

For example, if the two guards that were paired started with 3 and 5
bananas, after the first round of thumb wrestling they will have 6 and
2 (the one with 3 bananas wins and gets 3 bananas from the loser).
After the second round, they will have 4 and 4 (the one with 6 bananas
loses 2 bananas). At that point they stop and get back to guarding.

How is all this useful to distract the guards? Notice that if the
guards had started with 1 and 4 bananas, then they keep thumb
wrestling! 1, 4 -> 2, 3 -> 4, 1 -> 3, 2 -> 1, 4 and so on.

Now your plan is clear. You must pair up the guards in such a way that
the maximum number of guards go into an infinite thumb wrestling loop!

Write a function solution(banana_list) which, given a list of positive
integers depicting the amount of bananas the each guard starts with,
returns the fewest possible number of guards that will be left to
watch the prisoners. Element i of the list will be the number of
bananas that guard i (counting from 0) starts with.

The number of guards will be at least 1 and not more than 100, and the
number of bananas each guard starts with will be a positive integer no
more than 1073741823 (i.e. 2^30 -1). Some of them stockpile a LOT of
bananas.
```

To solve the problem, we first build a graph where each edge indicates a deadlock between two guards. We store this information in a matrix, where a 1 at position `(i,j)` represents a deadlock between guard `i` and guard `j`. A 0 represents a terminating match.

1. Set visited and path to empty arrays
2. The current vertex is popped of our list of unmatched vertices
3. Check if we can find an alternating extension of the current augmenting path
4. If we found an alternating extension, we add the current vertex to the path and move to the neighbor
5. In case we did not find an alternating extension, we backtrace
6. Now, if the current vertex (which is either the neighbor or the previous vertex) is in the list of unmatched vertices, we are going to add the vertex to our path and update our lists of matching and unmatching edges

We break the loop once we cannot find an augmenting_path. By subtracting the number of matches from the number of guards, we obtain the number of unmatched guards.

You can find the full problem and solution here.

Problem 4b.

I almost don't remember the second question of level 4.

```
Problem:

You need to free the bunny prisoners before Commander Lambda's space
station explodes! Unfortunately, the commander was
very careful with her highest-value prisoners - they're all held in
separate, maximum-security cells.
The cells are opened by putting keys into each console, then pressing
the open button on each console simultaneously.
When the open button is pressed, each key opens its corresponding lock
on the cell. So, the union of the keys in all of
the consoles must be all of the keys. The scheme may require multiple
copies of one key given to different minions.

The consoles are far enough apart that a separate minion is needed for
each one. Fortunately, you have already freed
```

```
just stealing all of the keys and using them
blindly. There are signs by the consoles saying how many minions had
some keys for the set of consoles.
You suspect that Commander Lambda has a systematic way to decide which
keys to give to each minion such that they
could use the consoles.

You need to figure out the scheme that Commander Lambda used to
distribute the keys. You know how many minions had keys,
and how many consoles are by each cell. You know that Command Lambda
wouldn't issue more keys than necessary
(beyond what the key distribution scheme requires), and that you need
as many bunnies with keys as there are consoles
to open the cell.

Given the number of bunnies available and the number of locks required
to open a cell, write a function
solution(num_buns, num_required) which returns a specification of how
to distribute the keys such that any num_required
bunnies can open the locks, but no group of (num_required - 1) bunnies
can.

Each lock is numbered starting from 0. The keys are numbered the same
as the lock they open (so for a duplicate key,
the number will repeat, since it opens the same lock). For a given
bunny, the keys they get is represented as a
sorted list of the numbers for the keys. To cover all of the bunnies,
the final answer is represented by a sorted
list of each individual bunny's list of keys.  Find the
lexicographically least such key distribution - that is,
the first bunny should have keys sequentially starting from 0.
```

The one thing I remember is that I found the problem very confusing when I read it. It took me some time to really understand what we are supposed to find out. Nevertheless, once the problem statement is clear, the solution is a simple combinatorics application (more precisely combinations).

```
from itertools import combinations

def solution(num_buns, num_required):
"""
```

```
    (num_required - 1)" bunnies,
    we would be able to open the door.

    => every one of the remaining "num_buns - num_required + 1" bunnies
    has exactly one key that
       is not in the union of keys of the selected "num_required - 1".

    This means that every key has exactly
    num_buns - num_required + 1
    copies (called copies_per_key)

    Total keys:
    -----------
    Based on the above logic, there are exactly
    /      num_buns     \
    \ num_required - 1 /
    different sets of bunnies that miss one key to being able to open the
    door.

    => there is a total number of distinct keys =
    /      num_buns     \
    \ num_required - 1 /
    =
    /            num_buns           \
    \ num_buns - num_required + 1 /
    = len(combinations(range(num_buns), copies_per_key))
    """

    key_sets = [[] for _ in range(num_buns)]

    copies_per_key = num_buns - num_required + 1

    for key, bunnies in enumerate(combinations(range(num_buns),
    copies_per_key)):
        for bunny in bunnies:
            key_sets[bunny].append(key)

    return key_sets
```

You can find the full problem and solution here.

Problem 5.

The problem is from the domain of group theory, which I quite enjoyed during university.

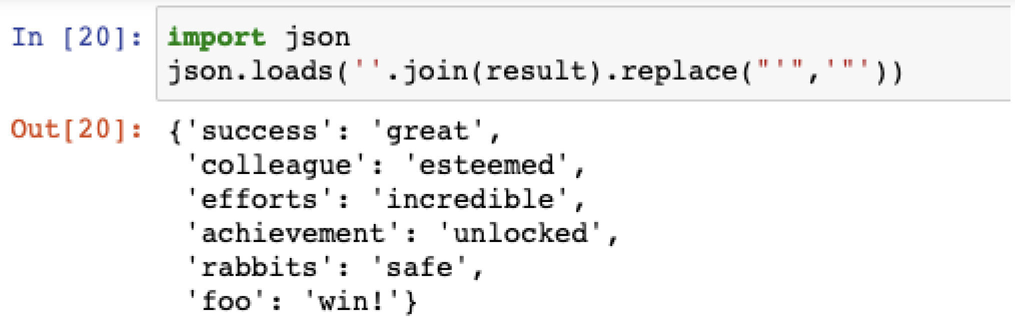
straight up copied the solution and referenced the original author as I felt like I couldn’t improve on the algorithm anyways.

```
Problem:

...

There's something important to note about quasar quantum flux fields'
configurations: when drawn on a star grid, configurations are
considered equivalent by grouping rather than by order. That is, for a
given set of configurations, if you exchange the position of any two
columns or any two rows some number of times, youll find that all of
those configurations are equivalent in that way - in grouping, rather
than order.

Write a function solution(w, h, s) that takes 3 integers and returns
the number of unique, non-equivalent configurations that can be found
on a star grid w blocks wide and h blocks tall where each celestial
body has s possible states. Equivalency is defined as above: any two
star grids with each celestial body in the same state where the actual
order of the rows and columns do not matter (and can thus be freely
swapped around). Star grid standardization means that the width and
height of the grid will always be between 1 and 12, inclusive. And
while there are a variety of celestial bodies in each grid, the number
of states of those bodies is between 2 and 20, inclusive. The solution
can be over 20 digits long, so return it as a decimal string.  The
intermediate values can also be large, so you will likely need to use
at least 64-bit integers.
```

You can find the full problem and solution here.

So what happened after I submitted Google’s Foobar?

Right after level 5, a base64 encoded string is shown. Decoding this string and then applying a bitwise exclusive or with my personal username and then converting the result back to Unicode lead to what I assume is some form of the final score.



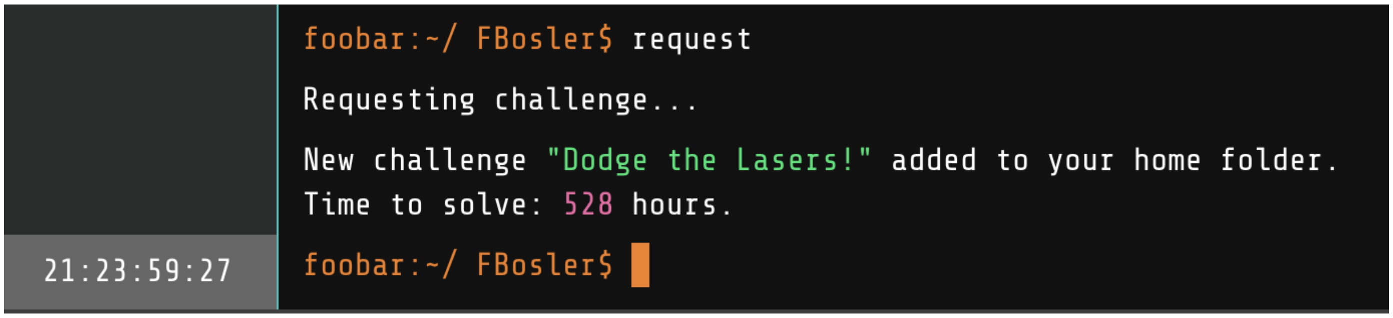
```
In [20]: import json
         json.loads(''.join(result).replace("'",'"'))

Out[20]: {'success': 'great',
          'colleague': 'esteemed',
          'efforts': 'incredible',
          'achievement': 'unlocked',
          'rabbits': 'safe',
          'foo': 'win!'}
```

Final deciphered message from Foobar

Well, this is a bit of a bummer. But then nothing happened. From what I read, sometimes a recruiter reaches out to the developer invited into Foobar. That did not happen here. I would have been rather curious to see what follows next.

Maybe I shouldn't have cheated on the last question 😂.

It might also have been because my code failed some internal test cases, or maybe it's because my CV doesn't fit the typical Google applicant. Either way, I had tons of fun with the challenge and just realized there are new questions in the Foobar server. My log-in still works, so I am going to solve some more problems! Stay tuned for more.



```
foobar:~/ FBosler$ request
Requesting challenge...
New challenge "Dodge the Lasers!" added to your home folder.
Time to solve: 528 hours.
foobar:~/ FBosler$
```

I just requested a new question. The clock is ticking again.

Update: Solved the new question. I also really like that one. Check out the corresponding article:

Dodge The Lasers — Fantastic Question From Google's hiring challenge

Thanks to Anupam Chugh